**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 5 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-12 Васильєв Єгор*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc52291748)

[2 Завдання 4](#_Toc52291749)

[3 Виконання 10](#_Toc52291750)

[3.1 Покроковий алгоритм 10](#_Toc52291751)

[3.2 Програмна реалізація алгоритму 10](#_Toc52291752)

[3.2.1 Вихідний код 10](#_Toc52291753)

[3.2.2 Приклади роботи 10](#_Toc52291754)

[3.3 Тестування алгоритму 11](#_Toc52291755)

[Висновок 12](#_Toc52291756)

[Критерії оцінювання 13](#_Toc52291757)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

# Завдання

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

* обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);
* зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;
* після цього параметр фіксується і змінюються інші параметри;
* далі повторюємо процедуру спочатку, з першого зафіксованого параметру;
* зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

|  |  |
| --- | --- |
| **№** | **Задача** |
| 1 | **Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.  Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика. |
| 2 | **Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.  **Розглядається симетричний, асиметричний та змішаний варіанти.**  В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.  У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.  У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.  Застосування:   * доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); * доставка води; * моніторинг об'єктів; * поповнення банкоматів готівкою; * збір співробітників для доставки вахтовим методом. |
| 3 | **Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число.  Застосування:   * розкладу для освітніх установ; * розкладу в спорті; * планування зустрічей, зборів, інтерв'ю; * розклади транспорту, в тому числі - авіатранспорту; * розкладу для комунальних служб; |
| 4 | **Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S.  Задача вершинного покриттяполягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).  На вході: Граф G = (V, E).  Результат: множина C ⊆ V - найменше вершинне покриття графа G.    Застосування:   * розміщення пунктів обслуговування; * призначення екіпажів на транспорт; * проектування інтегральних схем і конвеєрних ліній. |
| 5 | **Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній.  Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).  Застосування:   * біоінформатика; * електротехніка; |
| 6 | **Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.  Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

|  |  |
| --- | --- |
| **№** | **Алгоритми і досліджувані параметри** |
| 1 | **Генетичний алгоритм:**   * оператор схрещування (мінімум 3); * мутація (мінімум 2); * оператор локального покращення (мінімум 2). |
| 2 | **Мурашиний алгоритм**:   * α; * β; * ρ; * Lmin; * кількість мурах М і їх типи (елітні, тощо…); * маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**   * кількість ділянок; * кількість бджіл (фуражирів і розвідників). |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 1 | Задача про рюкзак + Генетичний алгоритм |
| 2 | Задача про рюкзак + Бджолиний алгоритм |
| 3 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 4 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 5 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 6 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 7 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 8 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |
| 9 | Задача вершинного покриття + Генетичний алгоритм |
| 10 | Задача вершинного покриття + Бджолиний алгоритм |
| 11 | Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм |
| 12 | Задача комівояжера (симетрична мережа) + Бджолиний алгоритм |
| 13 | Задача комівояжера (змішана мережа) + Бджолиний алгоритм |
| 14 | Розфарбовування графа + Генетичний алгоритм |
| 15 | Розфарбовування графа + Бджолиний алгоритм |
| 16 | Задача про кліку (задача розпізнавання) + Генетичний алгоритм |
| 17 | Задача про кліку (задача розпізнавання) + Бджолиний алгоритм |
| 18 | Задача про кліку (обчислювальна задача) + Генетичний алгоритм |
| 19 | Задача про кліку (обчислювальна задача) + Бджолиний алгоритм |
| 20 | Задача про найкоротший шлях + Генетичний алгоритм |
| 21 | Задача про найкоротший шлях + Мурашиний алгоритм |
| 22 | Задача про найкоротший шлях + Бджолиний алгоритм |
| 23 | Задача про рюкзак + Генетичний алгоритм |
| 24 | Задача про рюкзак + Бджолиний алгоритм |
| 25 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 26 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 27 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 28 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 29 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 30 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |

# Виконання

Варіант 7

## Покроковий алгоритм

ініціалізація початкових констант

def initialize():

ініціалізація матриці відстаней

ініціалізація списку доступних міст

ініціалізація матриці початкової концентрації феромона

ініціалізація матриці видимості

probabilities()

def get\_probabilities():

**для** i **в діапазоні**(0, кількість вершин):

**для** j **в діапазоні** (0, кількість вершин \_nodes):

numerator=pheromone\_matrix[i][j]\*\*ALPHA\*

\*self.visibility\_matrix[i][j] \*\*BETA

denominator = 0

**для** k **в діапазоні**(0, кількість вершин \_nodes):

denominator +=pheromone\_matrix[i][k] \*\*ALPHA \*

self.visibility\_matrix[i][k] \*\*BETA

self.probability\_matrix[i][j] = numerator / denominator

def choose\_next\_node(from\_node, ant\_type

probabilities = self.probability\_matrix[from\_node,

self.available\_nodes]

**якщо** ant\_type == "default":

probabilities /= sum(probabilities)

next\_node = np.random.choice(range(len(probabilities)),

p=probabilities)

**інакше**:

next\_node = np.random.choice(range(len(probabilities)))

**повернути** next\_node

def evaluate\_distance(paths):

best = float('inf')

scores = np.zeros(len(paths))

**для** index, path **в перелічених**(paths):

score = 0

**для** i **в діапазоні**(0, len(path) - 1):

score += map[path[i], path[i + 1]]

scores[index] = score

best = np.argmin(scores)

повернути paths[best], scores[best], scores

def evaporation():

self.pheromone\_matrix \*= (1 - self.EVAPORATION\_RATE)

def intensify(scores, paths):

i = self.best\_path[:-1]

j = self.best\_path[1:]

pheromone\_matrix[i, j] +=ELITE\_ANTS\_AMOUNT \* (L\_MIN /best\_score)

**для** index, score **в перелічених**(scores):

i = paths[index][:-1]

j = paths[index][1:]

pheromone\_matrix[i, j] += L\_MIN

def fit(map\_matrix):

map = map\_matrix

initialize()

num\_equal = 0

for i in range(ITERATION\_LIMIT):

start\_iter = time.time()

all\_paths = []

ant\_type = "default"

**для** ant **в діапазоні**(ANTS\_AMOUNT + WILD\_ANTS\_AMOUNT):

**якщо** ant >= ANTS\_AMOUNT:

ant\_type = "wild"

path = []

**якщо** START\_FROM\_DIFFERENT\_POINTS:

current\_node = available\_nodes[np.random.randint(0,

len(self.available\_nodes))]

**інакше**:

current\_node = self.available\_nodes[0]

start\_node = current\_node

**поки** True:

path.append(current\_node)

available\_nodes.remove(current\_node)

**якщо** len(self.available\_nodes) != 0:

current\_node\_index = self.choose\_next\_node(current\_node,

ant\_type)

current\_node = self.available\_nodes[current\_node\_index]

**інакше**:

**вихід з циклу**

path.append(start\_node)

available\_nodes = list(range(map.shape[0]))

all\_paths.append(path)

best\_path, best\_score, all\_scores =evaluate\_distance(all\_paths)

**якщо** i == 0:

self.best\_score = best\_score

self.best\_path = best\_path

**інакше**:

**якщо** best\_score < self.best\_score:

self.best\_score = best\_score

self.best\_path = best\_path

**якщо** best\_score == self.best\_score:

num\_equal += 1

**інакше**:

num\_equal = 0

all\_best\_scores.append(best\_score)

evaporation()

intensify(all\_scores, all\_paths)

get\_probabilities()

return self.best\_score

## Програмна реалізація алгоритму

### Вихідний код

import time

import matplotlib.pyplot as plt

import numpy as np

class AntColony:

ITERATION\_LIMIT = 100

ANTS\_AMOUNT = 10

ELITE\_ANTS\_AMOUNT = 2

WILD\_ANTS\_AMOUNT = 2

EVAPORATION\_RATE = 0.2

ALPHA = 2

BETA = 3

L\_MIN = 80

EARLY\_STOPPING\_COUNT = round(ITERATION\_LIMIT \* 0.7)

START\_FROM\_DIFFERENT\_POINTS = False

def \_\_init\_\_(self):

# internal representations

self.pheromone\_matrix = None

self.visibility\_matrix = None

self.probability\_matrix = None

self.map = None

self.available\_nodes = None

self.fit\_time = None

self.stopped\_early = False

# internal stats

self.best\_score = float('inf')

self.best\_path = None

self.all\_best\_scores = []

def initialize(self):

"""Initializes the model by creating the various matrices and generating the list of available nodes"""

num\_nodes = self.map.shape[0]

self.available\_nodes = list(range(num\_nodes))

self.pheromone\_matrix = np.zeros((num\_nodes, num\_nodes))

# remove the diagonal since there is no pheromone from node i to itself

self.pheromone\_matrix[np.eye(num\_nodes) == 0] = 0.1

self.visibility\_matrix = 1 / self.map

self.probability\_matrix = np.zeros((num\_nodes, num\_nodes))

self.get\_probabilities()

def get\_probabilities(self):

"""Obtaining a matrix of transition probabilities to a neighboring vertex"""

num\_nodes = self.map.shape[0]

for i in range(num\_nodes):

for j in range(num\_nodes):

numerator = self.pheromone\_matrix[i][j] \*\* self.ALPHA \* self.visibility\_matrix[i][j] \*\* self.BETA

denominator = 0

for k in range(num\_nodes):

denominator += self.pheromone\_matrix[i][k] \*\* self.ALPHA \* self.visibility\_matrix[i][k] \*\* self.BETA

self.probability\_matrix[i][j] = numerator / denominator

def choose\_next\_node(self, from\_node, ant\_type="default"):

"""Chooses the next node based on probabilities

:param ant\_type: type of ant, can be elite or default

:param from\_node: the node the ant is coming from

:return: index of the node the ant is going to"""

probabilities = self.probability\_matrix[from\_node, self.available\_nodes]

if ant\_type == "default":

probabilities /= sum(probabilities)

next\_node = np.random.choice(range(len(probabilities)), p=probabilities)

else:

next\_node = np.random.choice(range(len(probabilities)))

return next\_node

def evaluate\_distance(self, paths):

"""Evaluates the solutions of the ants by adding up the distances between nodes.

:param paths: solutions from the ants

:return: x and y coordinates of the best path as a tuple, the best path, and the best score"""

best = float('inf')

scores = np.zeros(len(paths))

for index, path in enumerate(paths):

score = 0

for i in range(len(path) - 1):

score += self.map[path[i], path[i + 1]]

scores[index] = score

best = np.argmin(scores)

return paths[best], scores[best], scores

def evaporation(self):

"""Evaporate some pheromone as the inverse of the evaporation rate."""

self.pheromone\_matrix \*= (1 - self.EVAPORATION\_RATE)

def intensify(self, scores, paths):

"""Increase the pheromone on traveled paths, and on a better path if elite ants are present.

:param paths: all the paths taken by the ants

:param scores: all costs of traveled paths"""

i = self.best\_path[:-1]

j = self.best\_path[1:]

self.pheromone\_matrix[i, j] += self.ELITE\_ANTS\_AMOUNT \* (self.L\_MIN / self.best\_score)

for index, score in enumerate(scores):

i = paths[index][:-1]

j = paths[index][1:]

self.pheromone\_matrix[i, j] += self.L\_MIN / score

def fit(self, map\_matrix):

"""Fits the ACO to a specific map.

:param map\_matrix: Distance matrix or some other matrix with similar properties"""

print("Beginning ACO Optimization with {} iterations...".format(self.ITERATION\_LIMIT))

self.map = map\_matrix

start = time.time()

self.initialize()

num\_equal = 0

for i in range(self.ITERATION\_LIMIT):

start\_iter = time.time()

all\_paths = []

ant\_type = "default"

for ant in range(self.ANTS\_AMOUNT + self.WILD\_ANTS\_AMOUNT):

if ant >= self.ANTS\_AMOUNT:

ant\_type = "wild"

path = []

if self.START\_FROM\_DIFFERENT\_POINTS:

current\_node = self.available\_nodes[np.random.randint(0, len(self.available\_nodes))]

else:

current\_node = self.available\_nodes[0]

start\_node = current\_node

while True:

path.append(current\_node)

self.available\_nodes.remove(current\_node)

if len(self.available\_nodes) != 0:

current\_node\_index = self.choose\_next\_node(current\_node, ant\_type)

current\_node = self.available\_nodes[current\_node\_index]

else:

break

path.append(start\_node) # go back to start

self.available\_nodes = list(range(self.map.shape[0]))

all\_paths.append(path)

best\_path, best\_score, all\_scores = self.evaluate\_distance(all\_paths)

if i == 0:

self.best\_score = best\_score

self.best\_path = best\_path

else:

if best\_score < self.best\_score:

self.best\_score = best\_score

self.best\_path = best\_path

if best\_score == self.best\_score:

num\_equal += 1

else:

num\_equal = 0

self.all\_best\_scores.append(best\_score)

self.evaporation()

self.intensify(all\_scores, all\_paths)

self.get\_probabilities()

print("Best score at iteration {}: {}; overall: {} ({:.2f}s)".format(i, best\_score,

self.best\_score,

time.time() - start\_iter))

if best\_score == self.best\_score and num\_equal == self.EARLY\_STOPPING\_COUNT:

self.stopped\_early = True

print("Stopping early due to {} iterations of the same score.".format(self.EARLY\_STOPPING\_COUNT))

print(self.best\_path)

break

self.fit\_time = (time.time() - start)

print("ACO fitted. Time taken: {} seconds. Best score: {}. Best path{}".format(round(self.fit\_time, 2),

self.best\_score,

self.best\_path))

return self.best\_score

def plot(self):

"""

Plots the score over time after the model has been fitted.

:return: None if the model isn't fitted yet

"""

fig, ax = plt.subplots(figsize=(20, 15))

ax.plot(self.all\_best\_scores, label="Best Run")

ax.set\_xlabel("Iteration")

ax.set\_ylabel("Performance")

ax.text(.8, .6,

'Default ants: {}\nElite ants: {}\nWild ants: {}\nEvap Rate: {}\nAlpha: {}\nBeta: {}\n\nFit Time: {}s'.format(

self.ANTS\_AMOUNT, self.ELITE\_ANTS\_AMOUNT, self.WILD\_ANTS\_AMOUNT, self.EVAPORATION\_RATE, self.ALPHA,

self.BETA, round(self.fit\_time, 2), ["\nStopped Early!" if self.stopped\_early else ""][0]),

bbox={'facecolor': 'gray', 'alpha': 0.8, 'pad': 10}, transform=ax.transAxes)

ax.legend()

plt.title("Ant Colony Optimization Results (best: {})".format(self.best\_score))

plt.show()

problem = np.random.randint(1, 50, size=(30, 30))

optimizer = AntColony()

best = optimizer.fit(problem)

optimizer.plot()

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

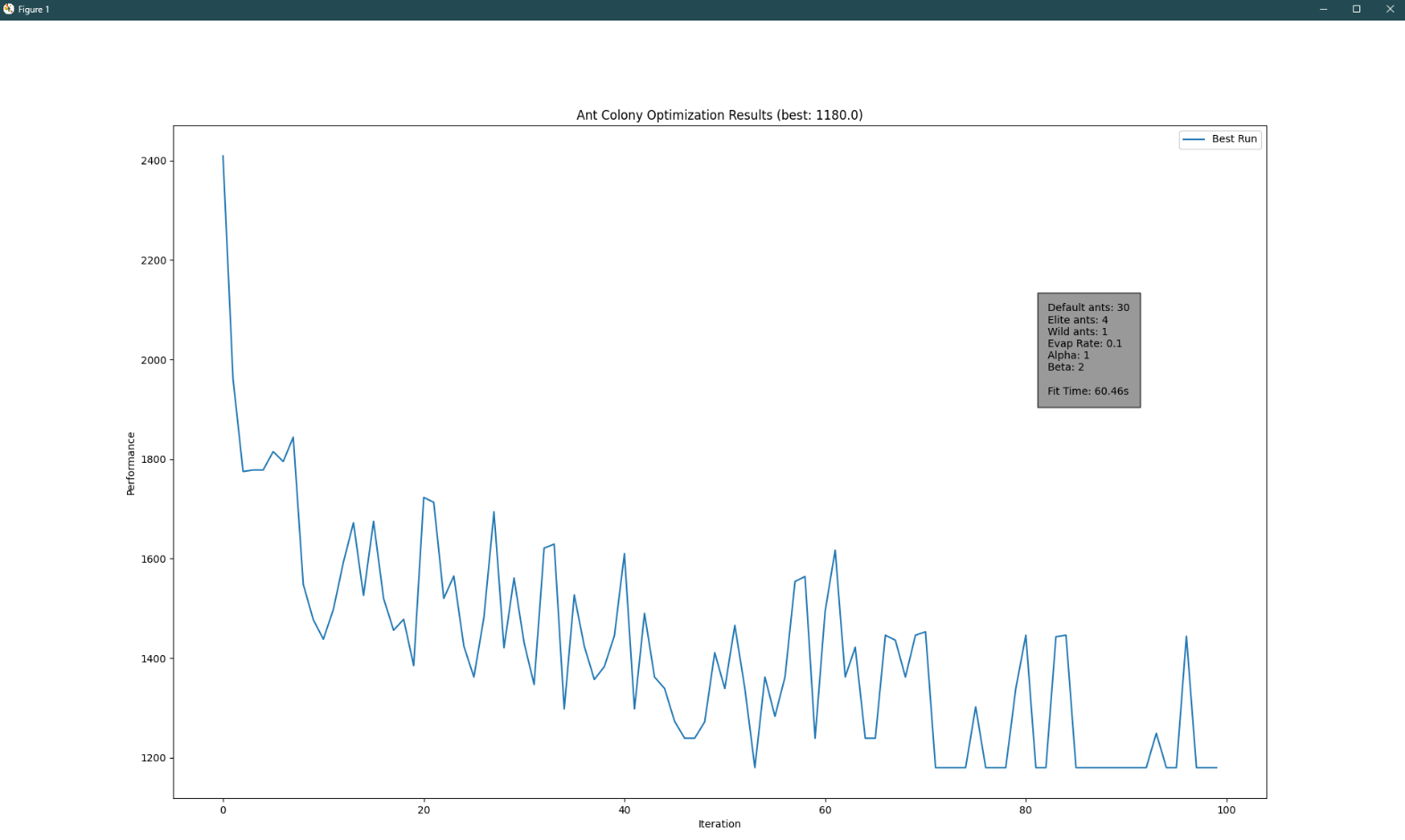


Рисунок 3.1 – Графік залежності загальної довжини маршруту від кількості ітерацій для 70 міст з випадковою відстанню між ними в межах від 1 до 500

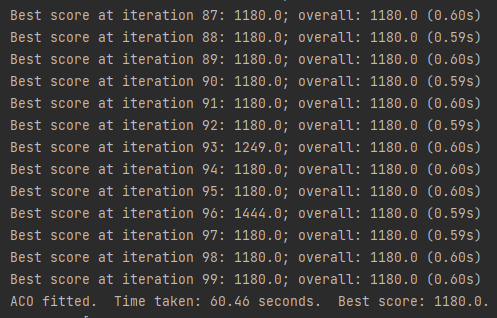


Рисунок 3.2 – Вивід інформації про розв’язок у консоль

## Тестування алгоритму

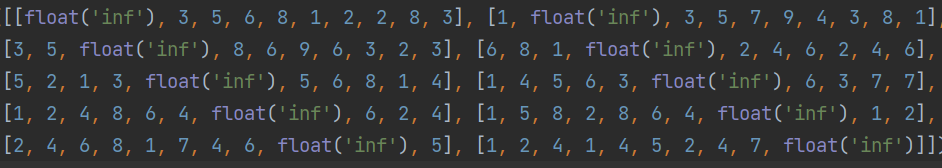
Для визначення оптимальних налаштувань, а саме коефіцієнтів α, β, , ρ, кількості звичайних, елітних (залишають більше феромону на кращому шляху) та диких мурах (обирають шлях абсолютно випадково, не зважаючи на вірогідності), а також початку маршрутів з різних точок, чи однієї, будемо фіксувати усі з них крім одного, та проводити декілька тестів на сталій матриці (рисунок 3.3). Результати тестування наведені в таблиці 3.1

Рисунок 3.3 – Матриця відстаней для тестування алгоритму

Таблиця 3.1 – Тестування алгоритму

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| α | β | ρ |  | Звичайні мурахи | Елітні мурахи | Дикі мурахи | Початок маршруту з | Загальна довжина кращого шляху |
| 2 | 3 | 0.2 | 20 | 5 | 2 | 1 | різних точок | 18 |
| 1 | 3 | 0.2 | 20 | 5 | 2 | 1 | різних точок | 16 |
| 3 | 3 | 0.2 | 20 | 5 | 2 | 1 | різних точок | 19 |
| 1 | 2 | 0.2 | 20 | 5 | 2 | 1 | різних точок | 18 |
| 1 | 4 | 0.2 | 20 | 5 | 2 | 1 | різних точок | 18 |
| 1 | 3 | 0.1 | 20 | 5 | 2 | 1 | різних точок | 18 |
| 1 | 3 | 0.3 | 20 | 5 | 2 | 1 | різних точок | 16 |
| 1 | 3 | 0.2 | 25 | 5 | 2 | 1 | різних точок | 18 |
| 1 | 3 | 0.2 | 15 | 5 | 2 | 1 | різних точок | 16 |
| 1 | 3 | 0.2 | 15 | 3 | 2 | 1 | різних точок | 18 |
| 1 | 3 | 0.2 | 15 | 8 | 2 | 1 | різних точок | 16 |
| 1 | 3 | 0.2 | 15 | 8 | 3 | 1 | різних точок | 16 |
| 1 | 3 | 0.2 | 15 | 8 | 1 | 1 | різних точок | 16 |
| 1 | 3 | 0.2 | 15 | 8 | 2 | 0 | різних точок | 16 |
| 1 | 3 | 0.2 | 15 | 8 | 2 | 2 | різних точок | 16 |
| 1 | 3 | 0.2 | 15 | 8 | 2 | 1 | однієї точки | 18 |

Висновок

В рамках даної лабораторної роботи було розроблено та виконано програмну реалізацію мурашиного алгоритму на прикладі задачі комівояжера з симетричною вежею. Даний алгоритм було детально протестовано на різній кількості міст та з різними діапазонами генерації випадкових відстаней між ними, після чого стало можливим стверджувати наступне:

* для коефіцієнтів α, β, ρ краще за все встановити фіксовані значення приблизно наступним чином: α=1, β=3, ρ=0.2.
* має бути трішки більше за найкращий прогнозований шлях, однак оскільки наперед оцінити такий шлях доволі складно, даний параметр краще визначати експериментально, поступово зменшуючи, починаючи з найгіршою оцінки до тих пір, поки довжина найкращого шляху буде зменшуватися.
* Кількість звичайних мурах краще залишати рівною кількості містам, оскільки при їх надлишковій кількості алгоритм буде виконуватися довше (хоча є ймовірність мінімального покращення кращого шляху), а при недостатній кількості розв’язок не буде оптимальним.
* Кількість елітних мурах краще встановити на рівні 10-20% від звичайних мурах, якщо брати більше, то мурахи будуть приділяти увагу лише локально кращому маршруту, не приділяючи достатньо уваги іншим, а якщо брати менше, то навпаки, будуть занадто далеко відходити від цього маршруту, та втратять можливість ще більшого скорочення локально кращого шляху.
* Дикі мурахи вносять хаотичний характер в пошук кращого маршруту, що, з одного боку, є корисним, оскільки з’являється шанс перевірити нові, на перший погляд не оптимальні шляхи, які можуть виявитись кращими, а з іншого боку, навпаки, роблять рішення імовірнісно-хаотичним, тому краще задавати їх кількість на рівні 20% від кількості елітних мурах.
* Починати завжди краще з різних, випадкових міст, що інтуїтивно зрозуміло та підтверджується на практиці.

Критерії оцінювання

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* покроковий алгоритм – 15%;
* програмна реалізація алгоритму – 50%;
* тестування алгоритму– 30%;
* висновок – 5%.